CG Programming I – Assignment #5 (Bézier Surface)

Due at the final.

In this assignment, a cubic Bézier surface will be rendered. The 16 control points for the surface will vary from frame to frame. Each frame, a set of matrices will be uploaded to the GPU. A vertex program will use these matrices to evaluate positions on the surface. In addition to evaluating the surface positions at each (u, v) value, the u and v tangents will also be calculated. These vectors will be used to further calculate the surface normal at each (u, v) value.

Part 1: Due on 28-November-2012 by the end of class

The first task is to derive the matrix form for evaluating a Bézier surface. This can be done in a similar manner as the matrix form for evaluating a Bézier curve. Recall the definition of a Bézier curve,

$$p(u) = \sum_{i=0}^{n} B_i^n(u) k_i \tag{1}$$

where n is the order of the curve and k_i represents one of the $n + 1$ control points. Equation (1) can be evaluated independently for each dimension. Rewriting equation (1) for independent evaluation in the x-, y-, and z-dimensions in $\mathbb{R}^3$ results in equation (2).

$$\begin{bmatrix} p_x(u) \\ p_y(u) \\ p_z(u) \end{bmatrix} = \begin{bmatrix} \sum_{i=0}^{3} B_i^3(u) k_{i,x} \\ \sum_{i=0}^{3} B_i^3(u) k_{i,y} \\ \sum_{i=0}^{3} B_i^3(u) k_{i,z} \end{bmatrix} \tag{2}$$

Each row in the vector on the right-hand side of equation (2) is a dot-product of $\begin{bmatrix} B_0^3 & B_1^3 & B_2^3 & B_3^3 \end{bmatrix}$ and the x-, y-, or z-components of the control points. This allows equation (2) to be rewritten as a matrix multiplication.

Let

$$\hat{u}^T = \begin{bmatrix} B_0^3(u) & B_1^3(u) & B_2^3(u) & B_3^3(u) \end{bmatrix}^T$$

and

$$K = \begin{bmatrix} k_{0,x} & k_{1,x} & k_{2,x} & k_{3,x} \\ k_{0,y} & k_{1,y} & k_{2,y} & k_{3,y} \\ k_{0,z} & k_{1,z} & k_{2,z} & k_{3,z} \end{bmatrix}$$

then

$$p(u) = K\hat{u} \tag{3}$$

for a cubic Bézier curve in $\mathbb{R}^3$.

The evaluation of a Bézier surface proceeds similarly. Instead of n control points, the Bézier surface has an $n \times m$ grid of control points. For a cubic Bézier surface, $n = m = 3$. Recall the definition of a Bézier surface,

$$p(u, v) = \sum_{i=0}^{n} \sum_{j=0}^{m} B_i^n(u) B_j^m(v) k_{i,j} \tag{4}$$

Using the distributive property, equation (4) can be rewritten as equation (5).

$$p(u, v) = \sum_{i=0}^{n} B_i^n(u) \left(\sum_{j=0}^{m} B_j^m(v) k_{i,j} \right) \tag{5}$$

Notice that the inner summation in equation (5) is just the evaluation of a Bézier curve. The n results of the inner summation is a set of n control points for another Bézier curve.

In a form similar to equation (3), derive a matrix form for evaluation of a Bézier surface. There are a few things to keep in mind

- Be aware of orientation and dimensions of vectors and matrices.
- Be aware of the indices, i and j, on the input control points.

- Where as equation (3) has a single matrix, K, as input, the equation for a Bézier surface should have four matrices as input.

Part 2: Due at the *start* of class 5-December-2012

To generate surface normals during the evaluation, the partial derivatives in the u and v directions must be evaluated.

$$\hat{n} = \left(\frac{\partial p(u,v)}{\partial u}\right) \times \left(\frac{\partial p(u,v)}{\partial v}\right)$$

These derivatives are shown in equations (6) and (7). Like in equation (5), the inner summation of each is just a Bézier curve that generates control points for another Bézier curve. In this case, however, the inner Bézier curve is one order lower than the original surface (i.e., quadratic instead of cubic).

$$\frac{\partial p(u,v)}{\partial u} = m \sum_{j=0}^{n} B_j^n(v) \left(\sum_{i=0}^{m-1} B_i^{m-1}(u) \left[p_{i+1,j} - p_{i,j} \right] \right) \tag{6}$$

$$\frac{\partial p(u,v)}{\partial v} = n \sum_{i=0}^{m} B_i^m(u) \left(\sum_{j=0}^{n-1} B_j^{n-1}(v) \left[p_{i,j+1} - p_{i,j} \right] \right) \tag{7}$$

Let

$$\hat{u}'^T = \begin{bmatrix} B_0^2(u) & B_1^2(u) & B_2^2(u) \end{bmatrix}^T$$

and

$$K' = \begin{bmatrix} k_{0,x} & k_{1,x} & k_{2,x} \\ k_{0,y} & k_{1,y} & k_{2,y} \\ k_{0,z} & k_{1,z} & k_{2,z} \end{bmatrix}$$

then

$$\frac{dp(u)}{du} = K' \hat{u}' \tag{8}$$

Equation (8) shows the matrix form for the evaluation of a quadratic Bézier curve in $\mathbb{R}^3$. Like in part 1, derive a matrix form solution for both $\frac{\partial p(u,v)}{\partial u}$ and $\frac{\partial p(u,v)}{\partial v}$.

Be prepared to hand in your equations at the beginning of class. We will discuss the solution first thing, so *late work will not be accepted.* You may also e-mail your solution to me before class. I will respond as quickly as I can. This will allow you to start on the next part soon, and that will be advantageous!

Part 3: Due at the final exam.

Using the equation defined in part 1, implement C++ code and vertex shader code to evaluate a Bézier surface. Per-frame control points are calculated (and clearly marked) in the supplied `main.cpp`. Modify the C++ to convert these control points to the required matrix form and supply this data the GPU shader. Implement a vertex shader in `bez-surf.vert` to evaluate the surface. The supplied `simple.vert` provides an example of accessing the (u, v) for each tessellated vertex in the patch. Code in `build_all_shaders` will need to be modified to load `bez-surf.vert` instead of `simple.vert`.

The matrices used in the surface evaluation will have 3 rows (because we're operating in $\mathbb{R}^3$) and 4 columns (because it is a cubic surface). The GLSL type for this kind of matrix is `mat4x3`. The surface evaluation shader should have an array of four of these matrices. For example,

```
uniform mat4x3 P[4];
```

declares a uniform variable `P` as an array of four `mat4x3`.

In order to supply data to the shader, the "location" of the uniform must be retrieved. This is somewhat like opening a file for write access. In `main.cpp` the `bezier_patch_program::get_uniform_locations` gets the locations for each of the uniforms in the shader. Code must be added to this function (and data members must be added to `bezier_patch_program`) to get and store the locations of any uniforms added to the shader.

For the simple vertex shader, a 3×3 matrix is used for the model-view matrix for the surface normal. Code in `Redisplay` prepares and uploads the data.

```
// Since the model-view matrix is an orthonormal basis, we can just
// use the upper 3x3 portion for transforming normals.
const float mv_normal[9] = {
    mv.col[0].values[0], mv.col[0].values[1], mv.col[0].values[2],
    mv.col[1].values[0], mv.col[1].values[1], mv.col[1].values[2],
    mv.col[2].values[0], mv.col[2].values[1], mv.col[2].values[2],
};
glUniformMatrix3fv(patch_program->mv_normal_uniform, 1, false, mv_normal);
```

Similar code will be needed to upload the matrices used for the surface evaluation. The array to store all of the data for these matrices will be 48 elements (12 elements for each matrix times 4 matrices). The upload of this data will use the function `glUniformMatrix4x3fv`. Like all functions in the `glUniform` family, the second parameter to this function is the number of array elements to upload. In this instance, "array elements" is the number GLSL array elements. The GLSL array is `mat4x3[4]`, and we want to upload all four elements.

Extra credit: Due at the final exam.

Using the matrices supplied to the vertex shader, calculate, *in the vertex shader*, the matrices used to calculate each of the partial derivatives. Using these matrices, calculate the partial derivatives and the surface normal. Change the value of the `DO_LIGHTING` define in `simple.frag` to perform lighting.

Debugging the calculation of normal vectors can be difficult. Since you can't single-step through a shader and inspect values, a different method is needed to visualize values that are generated by the code. A common way to do this is to emit a specific piece of data as the color. The fragment shader contains a debug define `SHOW_NORMALS`. If `DO_LIGHTING` is zero and `SHOW_NORMALS` is non-zero, the value of `normal_cs` in the fragment shader will be written as the color. If the vertex shader writes the object-space normal instead of the camera-space normal to `normal_cs`, you should be able to make some reasonable guess about what the normals should be in certain places at certain times.

For example, when the Bézier surface is flat, all of the normals should be $(0, 1, 0)$. Therefore, the colors should be $(.5, 1, .5)$ across the whole surface. The expected value is *not* $(0, 1, 0)$. Color values must be on the range $[0, 1]$, but normals can point in any direction, including negative directions. To visualize vectors as colors, the vectors have to be remapped from $[-1, 1]$ to $[0, 1]$. Zero in the $[-1, 1]$ space maps to .5 in the $[0, 1]$ space.

Criteria	Excellent	Good	Satisfactory	Unacceptable
Completion	Program correctly implements all required elements in a manner that is readily apparent when the program is executed. User interface is complete and responsive to input. Program documents user interface functionality.	Program implements all required elements, but some elements may not function correctly. User interface is complete and responsive to input.	Program implements most required elements. Some of the implemented elements may not function correctly. User interface is complete and responsive to input.	Many required elements are missing. User interface is incomplete or is not responsive to input.
Correctness	Program executes without errors. Program handles all special cases. Program contains error checking code.	Program executes without errors. Program handles most special cases.	Program executes without errors. Program handles some special cases.	Program does not execute due to errors. Little or no error checking code included.
Efficiency	Program uses solution that is easy to understand and maintain. Programmer has analysed many alternate solutions and has chosen the most efficient. Programmer has included the reasons for the solution chosen.	Program uses an efficient and easy to follow solution (i.e., no confusing tricks). Programmer has considered alternate solution and has chosen the most efficient.	Program uses a logical solution that is easy to follow, but it is not the most efficient. Programmer has considered alternate solutions.	Program uses a difficult and inefficient solution. Programmer has not considered alternate solutions.
Presentation & Organization	Program code is formatted in a consistent manner. Variables, functions, and data structures are named in a logical, consistent manner. Use of white space improves code readability.	Program code is formatted in mostly consistent with occasional inconsistencies. Variables, functions, and data structures are named in a logical, mostly consistent manner. Use of white space neither helps or hurts code reability.	Program code is formatted with multiple styles. Variables, functions, and data structures are named in a logical but inconsistent manner. Use of white space neither helps or hurts code reability.	Program code is formatted in an inconsistent manner. Variables, functions, and data structures are poorly named. Use of white space hurts code reability.
Documentation	Code clearly and effectively documented including descriptions of all global variables and all non-obvious local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted, as are the input requirements and output results.	Code documented including descriptions of most global variables and most non-obvious local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted, as are the input requirements and output results.	Code documented including descriptions of the most important global variables and the most important local variables. The specific purpose of each data type is noted. The specific purpose of each function is noted.	No useful documentation exists.

This rubric is based loosely on the "Rubric for the Assessment of Computer Programming" used by Queens University (http://educ.queensu.ca/ compsci/assessment/Bauman.html).